Impact of Coding Styles on Behaviours of Static Analysis Tools for Web Applications

Ibéria Medeiros, Nuno Neves

LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

imedeiros@di.fc.ul.pt, nuno@di.fc.ul.pt

I. INTRODUCTION

Web applications have become an essential resource to access the services of diverse subjects (e.g., financial, healthcare) available on the Internet. Despite the efforts that have been made on its security, namely on the investigation of better techniques to detect vulnerabilities on its source code, the number of vulnerabilities exploited has not decreased [1].

Static analysis tools (SATs) are often used to test the security of applications since their outcomes can help developers in the correction of the bugs they found [1]. There are SATs that only detect SQL injection (SQLi) and cross-site scripting (XSS) vulnerabilities [2], as they are the two most exploited [3], and others detect a few more classes of vulnerabilities [4][5]. However, the conducted investigation made over SATs stated they often generate errors (false positives (FP) and false negatives (FN)), whose cause is recurrently associated with very diverse coding styles, i.e., similar functionality is implemented in distinct manners, and programming practices that create ambiguity, such as the reuse and share of variables.

Based on a common practice of using multiple forms in a same webpage and its processing in a single file, we defined a use case for user login and register with six coding styles scenarios for processing their data, and evaluated the behaviour of three SATs (phpSAFE [2], RIPS [4] and WAP [5]) with them to verify and understand why SATs produce FP and FN.

II. USE CASE AND SCENARIOS

a) Multiple Forms Use Case: Multiple forms have been a practice used in current websites, e.g., for login and register a user (e.g., Facebook), but they are not limited to this. the HTML code that supports them is always the same, i.e., forms containing several input elements for designing the various pieces (e.g., buttons, input box, check box) they comprise. Another aspect that is common to multiple forms is that the server-side code (e.g., PHP) that receives and processes them is usually on the same file. Moreover, since only a form can be processed in turn, developers use the same variables to receive the user data coming from forms, and some parts of the code is common to all forms. Besides these practices, which are not incorrect, the way that the data processing is codded can differ among developers, but for the same purpose.

Listing 1 presents the PHP code for processing the user login and register operations. Variables $email and $pw will receive the common entry points of both forms. This means that they are the same for both forms but are processed

separately and in distinct operations (login and register). Also, variables $sql and $res will be (re)used along the file to compose queries and get their results.

```php
1 <?php
2
3 // common variables to both forms.
4 //$email = $_GET["email"];
5 //$pw = $_GET["email_pass"];
6
7 // code to process the login form
8 if (isset($_GET["login"])) {
9    $email = $_GET["email"];
10   $pw = $_GET["email_pass"];
11   $sql= "SELECT * FROM users WHERE addr ='".$email."'
          AND addr_pass ='".$pw."'";
12   $res = mysqli_query($con, $sql);
13 }
14
15 // code to process the register form
16 if (isset($_GET["register"])) {
17   $name = $_GET["name"];
18   $email = $_GET["email"];
19   $pw = $_GET["email_pass"];
20   $pw_conf = $_GET["email_pass_conf"];
21   $sql= "SELECT * FROM users WHERE addr ='".$email."'";
22   $res = mysqli_query($con, $sql);
23   if (mysqli_num_rows($res) != 0)
24     echo "This user already exist";
25   else{
26     if ($pw === $pw_conf){
27       $sql = "INSERT INTO users ('name', 'email',
              'password') VALUES ('".$name."',
              '".$email."', '".$pw."')";
28       $res = mysqli_query($con, $sql);
29     }else
30       echo "Passwords do not match";
31     }
32 }
33
34 // code to process the final query of both forms
35 //$res = mysqli_query($con, $sql);
36
37 // code to process the login form
38 //if (isset($_GET["login"])) {
39 // $email = $_GET["email"];
40 // $pw = $_GET["email_pass"];
41 // $sql= "SELECT * FROM users WHERE addr ='".$email."'
          AND addr_pass ='".$pw."'";
42 //}
43
44 // code to process the login form
45 //if (isset($_GET["login"])) {
46 // $res = mysqli_query($con, $sql);
47 // if (mysqli_num_rows($res) == 1)
48 //   echo "Login successful"
49 //}
50 ?>
```

Listing 1. PHP code for processing the user login and register forms.

b) Coding Style Scenarios: Based on the code of Listing 1, we defined six coding style scenarios for processing the data provided from login and register forms. For that, we
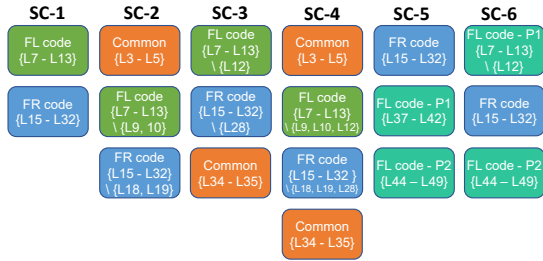
SC-1
- FL code {L7 - L13}
- FR code {L15 - L32}

SC-2
- Common {L3 - L5}
- FL code {L7 - L13} \ {L12}
- FR code {L15 - L32} \ {L18, L19}

SC-3
- FL code {L7 - L13} \ {L12}
- FR code {L15 - L32} \ {L28}
- Common {L34 - L35}

SC-4
- Common {L3 - L5}
- FL code {L7 - L13} \ {L9, L10, L12}
- FR code {L15 - L32} \ {L18, L19, L28}
- Common {L34 - L35}

SC-5
- FR code {L15 - L32}
- FL code - P1 {L37 - L42}
- FL code - P2 {L44 - L49}

SC-6
- FL code - P1 {L7 - L13} \ {L12}
- FR code {L15 - L32}
- FL code - P2 {L44 - L49}

Fig. 1. Considered scenarios to process multiple forms on a single PHP file.

identified different code blocks (`cb`), and then we built the scenarios. Figure 1 shows these scenarios with their code blocks, specifying the code lines for each of them (between braces). Specific `cb` for login operation are green, whereas `cb` for register operation are blue. Orange `cb` are those that are common on processing of both forms. The light green `cb` is the code of the login operation but split on two `cb` (P1 and P2). Analyzing the code of Listing 1, the first scenario (SC-1) is active. To enable the other scenarios, it needs to activate their lines, by uncommenting some lines and commenting others. Finally, the notation $\{Lx - Ly\} \setminus \{Lz\}$ means that the `cb` is defined by the $\{Lx-Ly\}$ range of lines of code, except the Lz line. For example, SC-2 is composed as follows: a common `cb` which is executed for both operations, a `cb` that is only executed for login operation, and a `cb` that is only executed for register operation. Each scenario contains three SQLi: one on the login operation and two on the register operation.

III. ANALYSIS OF SAT'S BEHAVIORS

We run the three tools over the six scenarios and we analysed their outcomes to understand their behaviours and check the veracity of their results. *All SATs had the same results and behaviours*. Table I presents the results.

a) SC-1 and SC-2: For these scenarios, *the tools correctly detected all vulnerabilities*. These results are justified by the facts of the code for login and register operations is well delimited in both scenarios and the entry points are used in a distinct sink, which only belongs to an operation. This can reduce SATs to incur in a wrong analysis. Although SC-2 has a common `cb`, containing the shared entry point for both operations, it does not affect the SATs's behaviour since the variables $email and $pw that receive these entries do not change (reassigned) along the `cb` of each operation.

TABLE I
RESULTS OF SATs OVER THE SIX CODDING STYLE SCENARIOS.

Scenario	Vulnerability	TP	FN	FP
SC-1	{L9, L10, L11, L12}	1		
	{L18, L21, L22}	1		
	{L17, L18, L19, L27, L28}	1		
SC-2	{L4, L5, L11, L12}	1		
	{L4, L21, L22}	1		
	{L4, L5, L17, L27, L28}	1		
SC-3	{L18, L21, L22}	1		
	{L9, L10, L11, L35}		1	
	{L17, L18, L19, L27, L35}	1		
SC-4	{L4, L21, L22}	1		
	{L4, L5, L11, L35}		1	
	{L4, L5, L17, L27, L35}	1		
SC-5	{L18, L21, L22}	1		
	{L17, L18, L19, L27, L28}	1		
	{L39, L40, L41, L46}	1		
SC-6	{L9, L10, L11, L46}		1	
	{L17, L18, L19, L27, L28}	1		
	{L18, L21, L22}	1		1: L46

TP: true positive; FN: false negative; FP: false positive.

b) SC-3 and SC-4: All tools correctly detected two vulnerabilities and had a FN. Both scenarios include common blocks. SC-4 has the same common `cb` as SC-2, which does not interfere in the SAT's analysis as we see above. In contrast, the common `cb` that ends both scenarios affects the analysis performed by tools. This `cb` contains a sink that receives two distinct sets of entry points, each one from each login and register operation. Also, the $sql variable is used in both operations. Since only one operation is expected to be executed, the programmer's decision of using these variables and sink the same way for both operation is correct. However, SATs do not have this knowledge, hence, this leads to FN. Therefore, SATs only detected the vulnerability whose $sql is assigned closer to the common sink (i.e., the second vulnerability of register operation), and generated a FN for the vulnerability where $sql is assigned farther from that sink.

c) SC-5: All tools detected the three vulnerabilities. Their outcomes are justified by the absence of common `cb`, the register operation code is well delimited and the login operation code is split on two blocks (P1 and P2), but as they are sequential, they work as a single block.

d) SC-6: This scenario was the one that tools had the worsts results. *The tools correctly detected two vulnerabilities and produced a FN and a FP in which it is an inexistent execution path*. The composition of SC-6 is similar to SC-5; however, P1 and P2 are placed, respectively, before and after the register operation block. P2 contains the sink that receives the query composed on P1. The vulnerability associated with the login operation is not detected (FN) and in its place is produced a FP. As SATs are not able to distinguish the code that belongs to each operation, their result can lead to false execution paths; so, FP and FN.

The results shows that SATs are built having in mind how to detect specific vulnerabilities, without considering the coding styles. Moreover, these styles underlie SATs errors, and for the best of our knowledge, their impact on SATs never was studied. These results are the outcome of a preliminary study that we conducted and they call for the action for a new generation of SAT tools.

REFERENCES

[1] WhiteHat Security, "The DevSecOps Approach - Using AppSec Statistics to Drive Better Outcomes," Nov. 2019.

[2] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2015.

[3] J. Williams and D. Wichers, "OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks," 2017.

[4] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[5] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the International World Wide Web Conference*, Apr. 2014, pp. 63–74.